PARALLEL SOFTWARE TECHNOLOGIES LABORATORY PSTL

Scalable and Modular Parallel I/O for Open MPI

Edgar Gabriel

Parallel Software Technologies Laboratory
Department of Computer Science,
University of Houston
gabriel@cs.uh.edu

CS@UH

Outline

- Motivation
- MPI I/O: basic concepts
- OMPIO module and parallel I/O frameworks in Open MPI
- Parallel I/O research
- Conclusions and future work

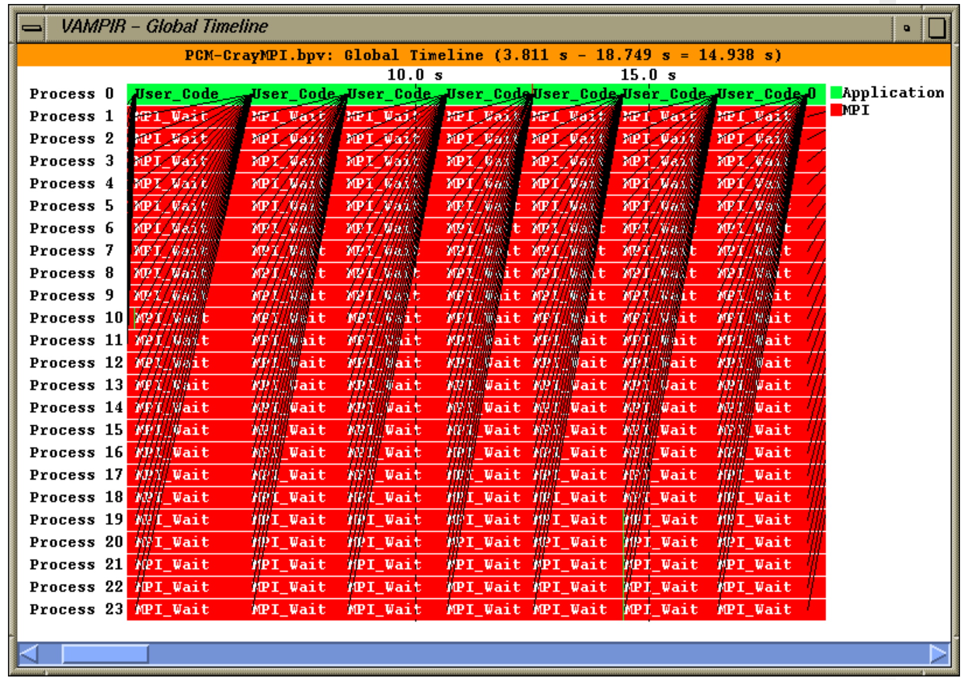
Motivation

- Study by LLNL (2005):
  - 1 GB/s I/O bandwidth required per Teraflop compute capability
  - Write to the filesystem dominates reading from it by a factor of 5
- Current High-End Systems:
  - K Computer: ~11 PFLOPS, ~96 GB/s I/O bandwidth using 864 OSTs
  - Jaguar (2010): ~1 PFLOPS, ~90 GB/s I/O bandwidth using 672 OSTs

⇨ Gap between available I/O performance and required I/O performance.

Application Perspective

- Sequential I/O
  - A single process executes file operations
  - Leads to load imbalance
- Individual I/O
  - Each process has its own files
  - Pre/Post-processing required



- Parallel I/O
  - Multiple processes access (different parts of) the same file (efficiently)

Part I: MPI I/O

MPI I/O

- MPI (Message Passing Interface) version 2 introduced the notion of parallel I/O
  - **Collective I/O** : group I/O operations
  - **File view**: registering an access plan to the file in advance
  - **Hints:** application hints on the lanned usage of the
  - **Relaxed consistency semantics**: updates to a file might initially only be visible to the process performing the action
  - **Non-blocking I/O:** asynchronous I/O operations

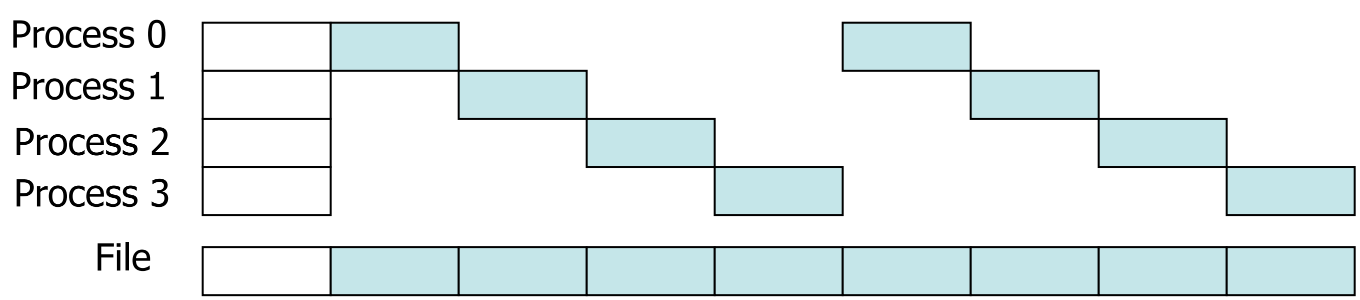
General file manipulation functions

```
MPI_File_open ( MPI_Comm comm, char *filename,
                int amode, MPI_Info info,
                MPI_File *fh );
```

- Collective operation
  - All processes have to provide the same `amode`
  - `comm` must be an intra-communicator
- Values for `amode`
  - `MPI_MODE_RDONLY, MPI_MODE_WRONLY, MPI_MODE_RDWR,`
  - `MPI_MODE_CREATE, MPI_MODE_APPEND, …`
- Combination of several `amodes` possible, e.g
  - C: `(MPI_MODE_CREATE | MPI_MODE_WRONLY)`
  - Fortran: `MPI_MODE_CREATE + MPI_MODE_WRONLY`

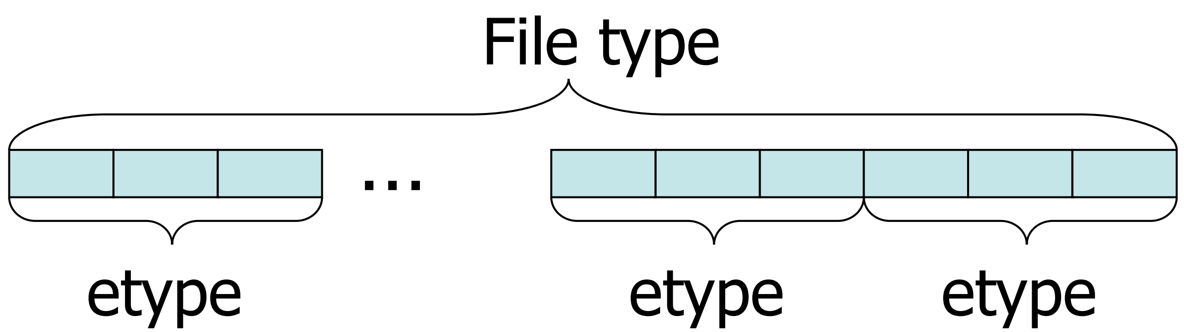
File View

- File view: portion of a file visible to a process
  - Processes can share a common view
  - Views can overlap or be disjoint
  - Views can be changed during runtime
  - A process can have multiple instances of a file open using different file views



File View

- Elementary type (etype) : basic unit of the data accessed by the program
- File type: datatype used to construct the file view
  - consists logically of a series of etypes
  - must not have overlapping regions if used in write operations
  - displacements must increase monotonically
- Default file view:
  - displacement = 0
  - etype = file type = `MPI_Byte`



Setting a file view

```
MPI_File_set_view ( MPI_File fh, MPI_Offset disp,
      MPI_Datatype etype, MPI_Datatype filetype,
      char *datarep, MPI_Info info );
```

- The argument list
  - `disp`: start of the file view
  - `etype` and `filetype`: as discussed previously
  - `datarep`: data representation used
  - `info`: hints to the MPI library (discussed later)
- Collective operation
  - `datarep` and extent of `etype` have to be identical on all processes
  - `filetype, disp` and `info` might vary
- Resets file pointers to zero

File Interoperability

- Fifth parameter of `MPI_File_set_view` sets the data representation used:
  - `native`: data is stored in a file exactly as it is in memory
  - `internal`: data representation for heterogeneous environments using the same MPI I/O implementation
  - `external32`: portable data representation across multiple platforms and MPI I/O libraries.

- User can register its own data representation, providing the according conversion functions (`MPI_Register_datarep`)

General file manipulation functions

```
MPI_File_read ( MPI_File fh, void *buf, int cnt,
               MPI_Datatype dat, MPI_Status *stat);
MPI_File_write ( MPI_File fh, void *buf, int cnt,
               MPI_Datatype dat, MPI_Status *stat);
```

- Buffers described by the tuple of
  - Buffer pointer
  - Number of elements
  - Datatype
- Interfaces support data conversion if necessary

MPI I/O non-collective functions

Positioning	Synchronism	Function
Individual file pointers	Blocking	`MPI_File_read`
		`MPI_File_write`
	Non-blocking	`MPI_File_iread`
		`MPI_File_iwrite`
Explicit offset	Blocking	`MPI_File_read_at`
		`MPI_File_write_at`
	Non-blocking	`MPI_File_iread_at`
		`MPI_File_iwrite_at`
Shared file pointers	Blocking	`MPI_File_read_shared`
		`MPI_File_write_shared`
	Non-blocking	`MPI_File_iread_shared`
		`MPI_File_iwrite_shared`

Individual I/O in parallel applications

1	**2**	**3**	**4**	5	6	7	8
9	**10**	**11**	**12**	13	14	15	16
17	**18**	**19**	**20**	21	22	23	24
25	**26**	**27**	**28**	29	30	31	32

Process 0:

```
read(…, offset=0,  length=2)
read(…, offset=8,  length=2)
read(…, offset=16, length=2)
read(…, offset=24, length=2)
```

- Individual Read/Write operations on a joint file often lead to many, small I/O requests from each process
- Arbitrary order of I/O requests from the file system perspective
  - Will lead to suboptimal performance

Collective I/O in parallel applications

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32

Process 0:

```
read(…, offset=0,  length=4)
MPI_Send (…,length=2,dest=3,…)
read(…, offset=8, length=4)
MPI_Send (…,length=2,dest=3,…)
read(…, offset=16, length=4)
MPI_Send (…,length=2,dest=3,…)
read(…, offset=24, length=4)
MPI_Send (…,length=2,dest=3,…)
```

- Collective I/O:
  - Offers the potential to rearrange I/O requests across processes, e.g. minimize file pointer movements, minimize locking occurring on the file system level
  - Offers performance benefits if costs of additional data movements < benefit of fewer repositioning of file pointers

Collective I/O: Two-phase I/O algorithm

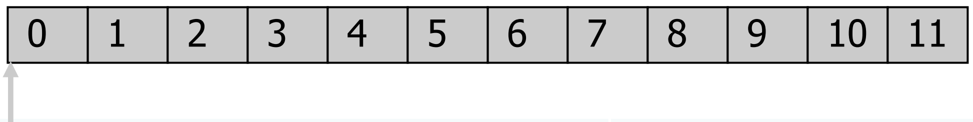
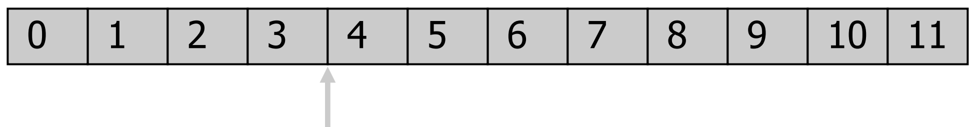
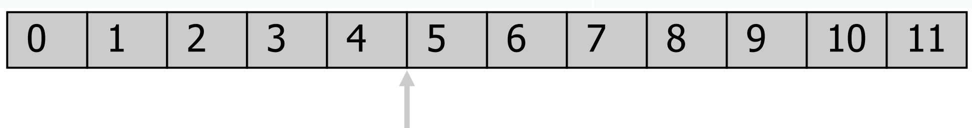
- Re-organize data across processes to match data layout in file
- Combination of I/O and (MPI level) communication used to read/write data from/to file
- Only a subset of processes actually touch the file (aggregators)
- Large read/write operations split into multiple cycles internally
  - Limits the size of temporary buffers
  - Overlaps communication and I/O operations

Shared File Pointer Operations

- Shared file pointers: a file pointer shared by a the group of processes that has been used to open the file
  - All processes must have identical file view
  - Might lead to non-deterministic behavior
- Shared file pointer must not interfere with the individual file pointer of each process

- Typical usage scenarios
  - Writing a parallel log-file
  - Work distribution across processes by reading data from a joint file

Shared file pointer example

<table>
<tr><th>Time step</th><th>Process 0</th><th>Process 1</th></tr>
<tr><td colspan="3">
</td></tr>
<tr><td>T1</td><td>MPI_File_read_shared (…, 4, MPI_INT,…)</td><td>-</td></tr>
<tr><td colspan="3">
</td></tr>
<tr><td>T2</td><td>-</td><td>MPI_File_read_shared (…, 1, MPI_INT,…)</td></tr>
<tr><td colspan="3">
</td></tr>
<tr><td>T3</td><td>MPI_File_read_sharedl(…, 1,MPI_INT,…)</td><td>MPI_File_read_sharedl(…,2,MPI_INT,)</td></tr>
<tr><td></td><td>?</td><td>?</td></tr>
</table>

Time step	Action by process 0	Action by process 1

Access to shared file pointer is serialized and execute either as…

Time step	Action by process 0	Action by process 1
T3a		MPI_File_read_shared (…, 2, MPI_INT,…)

0	1	2	3	4	5	6	7	8	9	10	11

Time step	Action by process 0	Action by process 1
T3b	MPI_File_read_shared (…, 1, MPI_INT,…)	

0	1	2	3	4	5	6	7	8	9	10	11

…or the other way around. Which ever process gets hold of the shared file pointer first is allowed to execute first the read operation

Time step	Action by process 0	Action by process 1
T3a	MPI_File_read_shared (…, 1, MPI_INT,…)	

0	1	2	3	4	5	6	7	8	9	10	11

Time step	Action by process 0	Action by process 1
T3b		MPI_File_read_shared (…, 2, MPI_INT,…)

0	1	2	3	4	5	6	7	8	9	10	11

Consistency of file operation

- MPI does not provide sequential consistency across all processes per default
  - Write on one process is initially just visible on the same process
- Two possibilities to change this behavior

```
MPI_File_set_atomicity ( MPI_File fh, int flag );
```

  - If flag = true, all write operations are atomic
  - Collective operation

```
MPI_File_sync ( MPI_File fh );
```

  - Flushes all write operations on the calling process' file instance

Hints supported by MPI I/O (I)

Hint	Explanation	Possible values
`access_style`	Specifies manner in which the file is accessed	`read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential, random`
`collective_buffering`	Use collective buffering ?	`true, false`
`cb_block_size`	Block size used for collective buffering	Integer
`cb_buffer_size`	Total buffer space that can be used for collective buffering	Integer, multiple of cb_block_size
`cb_nodes`	Number of target nodes used for collective buffering	Integer

Hints supported by MPI I/O (II)

Hint	Explanation	Possible values
`io_node_list`	List of I/O nodes that should be used	Comma separated list of strings
`nb_proc`	Specifies the number of processes typically accessing the file	Integer
`num_io_nodes`	Number of I/O nodes available in the system	Integer
`striping_factor`	Number of I/O nodes that should be used for file striping	Integer
`striping_unit`	Stripe depth	integer

Part II: OMPIO

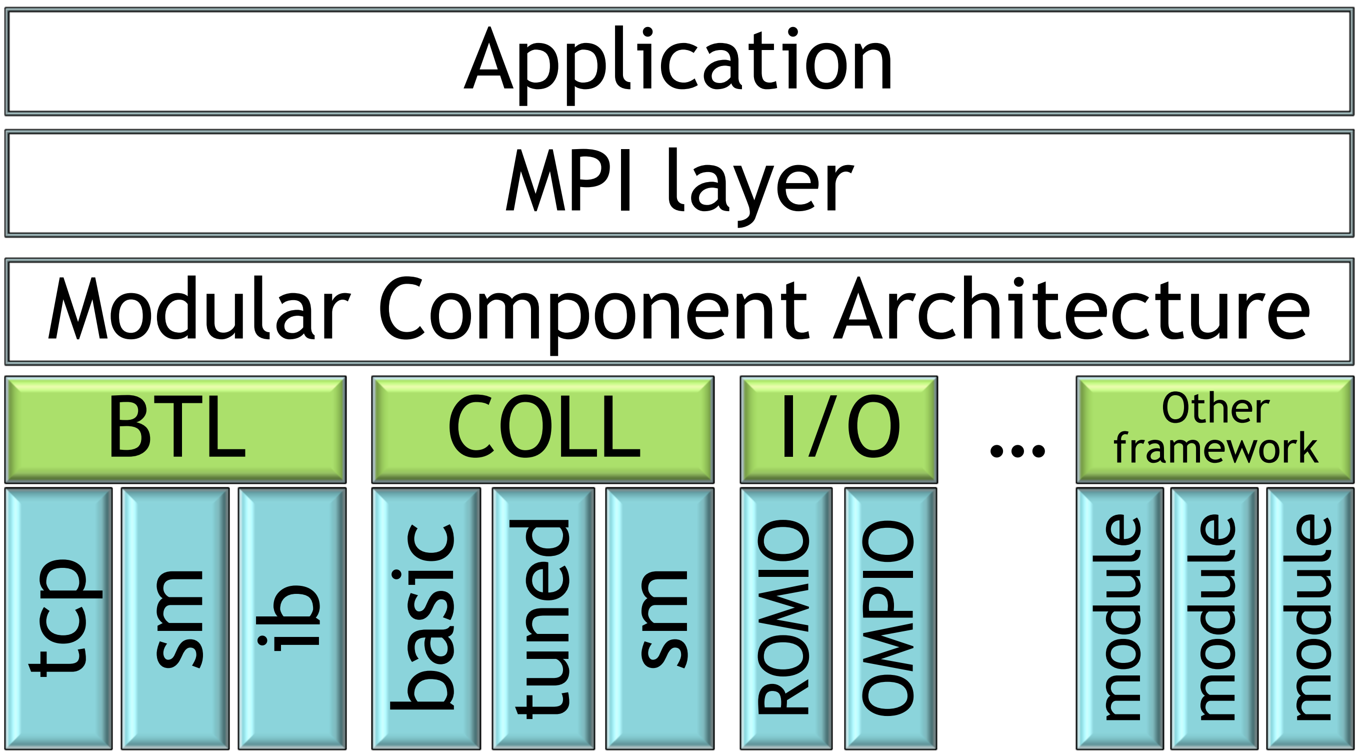
OMPIO Design Goals (I)

- Highly modular architecture for parallel I/O
  - Maximize code reuse, minimize code replication
- Generalize the selection of modules
  - Collective I/O algorithms
  - Shared file pointer operations
- Tighter Integration with Open MPI library
  - Derived data type optimizations
  - Progress engine for non-blocking I/O operations
  - External data representations etc.

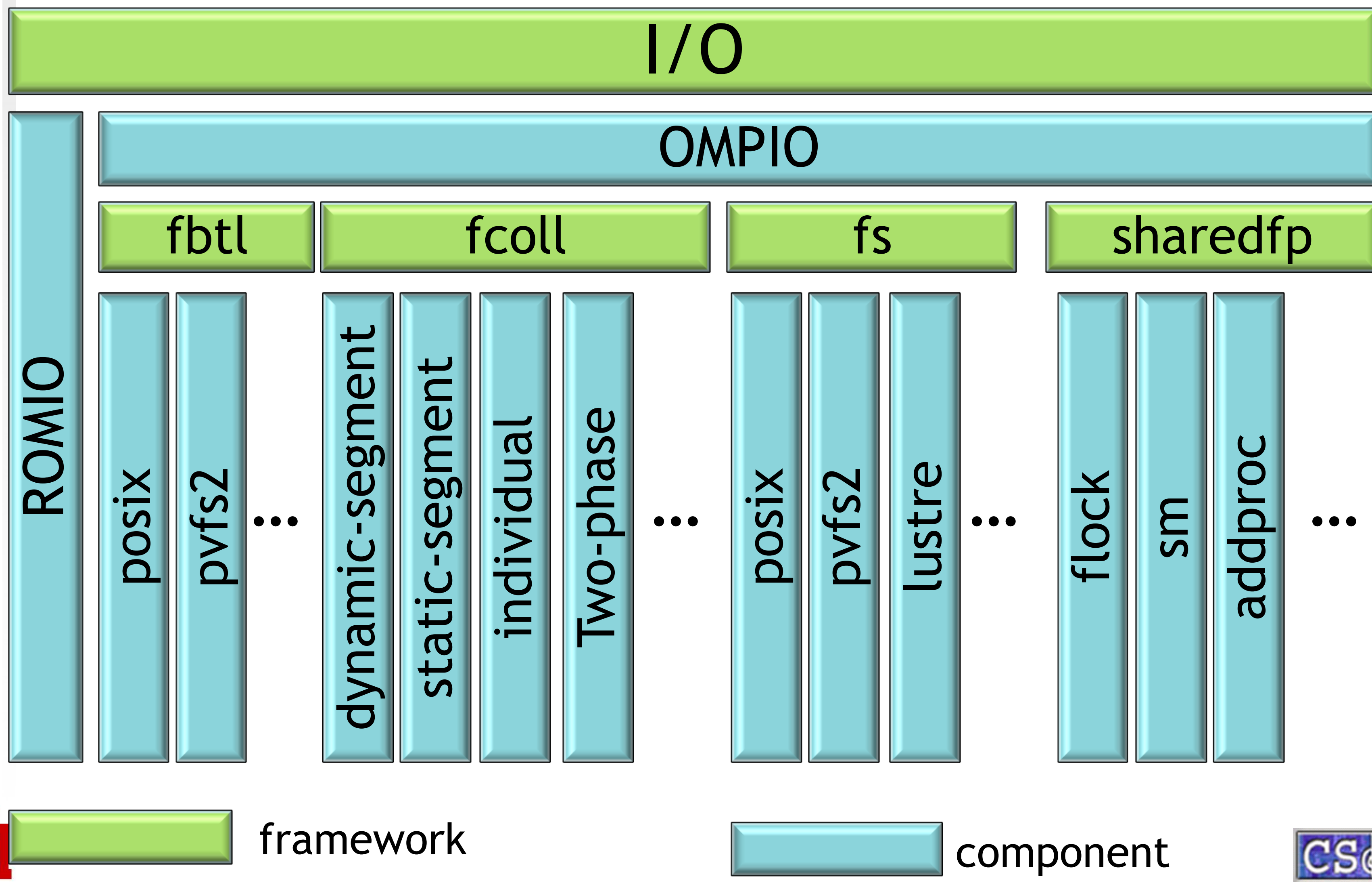
OMPIO Design Goals (II)

- Adaptability
  - Enormous diversity of I/O hardware and software solutions
    - Number of storage server, bandwidth of each storage server
    - Network connectivity in-between I/O nodes, between compute and I/O nodes, and message passing network between compute nodes
  - Ease the modification of module parameters
  - Ease the development and dropping in of new modules

Open MPI Architecture



OMPIO frameworks overview



OMPIO

- Main I/O component
- 'Understands' MPI semantics
- Translates MPI write/read operations into lower layer operations
- Provides the implementation and the operation of the
  - `MPI_File` handle
  - File view operations
  - (`MPI_Request` structures)
- Triggers upon selection the `fcoll, fs, fbtl` and `sharedfp` selection logic

fbtl: file byte transfer layer

- Abstraction for individual read and write operations
- A process will have per MPI file one or more fbtl modules loaded
- Main interfaces work with the tuple of `<buffer pointer, length, position in file>`

- Interface:
  - `pwritev()`
  - `preadv()`
  - `ipwritev()`
  - `ipreadv()`
  - `progress()`

PARALLEL SOFTWARE TECHNOLOGIES LABORATORY PSTL

fcoll: collective I/O framework

- Provides implementations of the collective I/O operations of the MPI specification

```
– read_all()         - read_all_begin()/end()
– write_all()        - write_all_begin()/end()
– read_at_all()      - read_at_all_begin()/end()
– write_at_all()     - write_at_all_begin()/end()
```

- Selection logic triggered upon setting the file view

fcoll: selection logic

- Decision between different collective modules based on:
  - `ss:` stripe size of the file system
  - `c:` average contiguous chunk size in file view
  - `k:` minimum data size to saturate write/read bandwidth from one process
  - `size of gap` in the file view between processes.

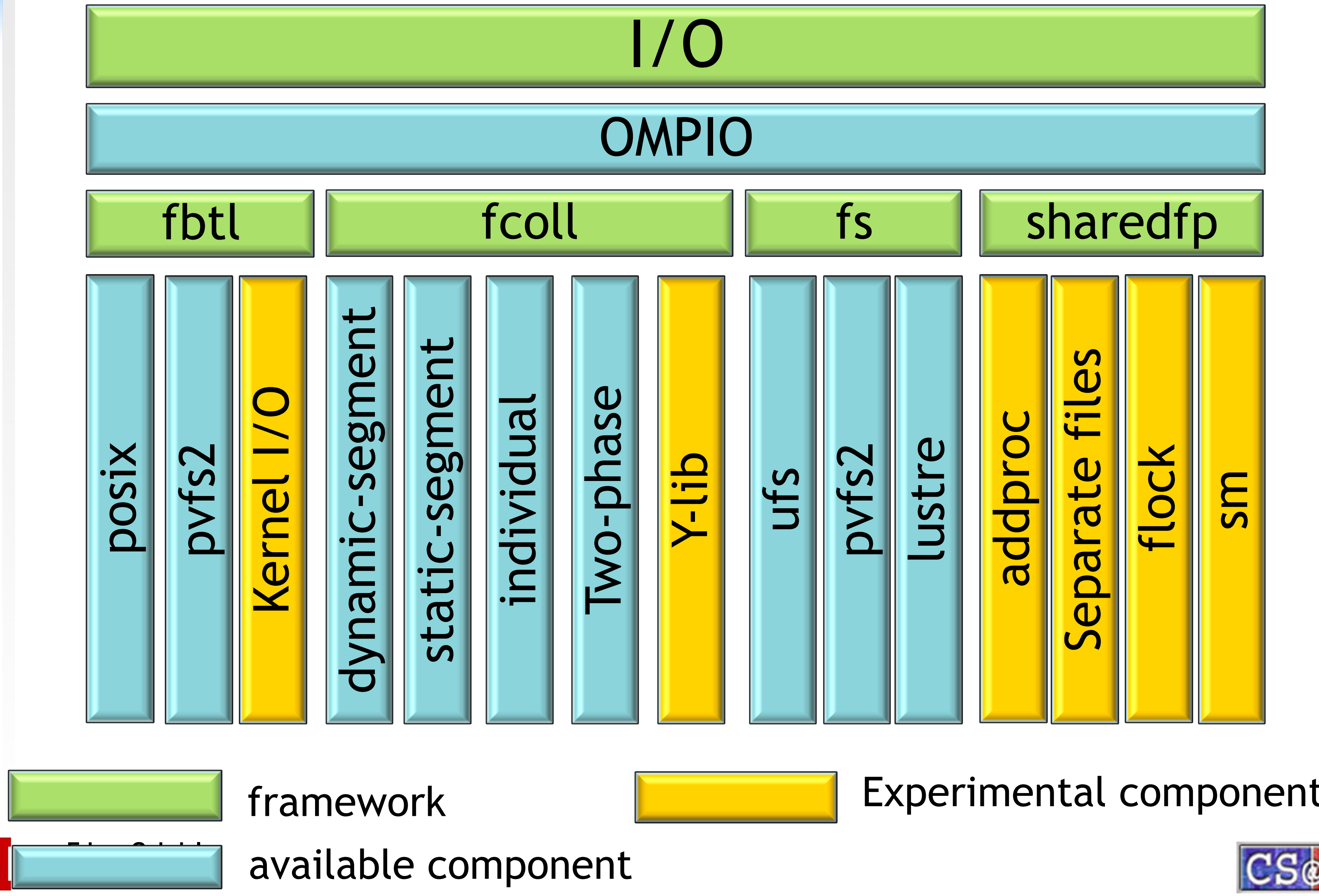
Characteristic	Gap Size	Algorithm
c>k and c>ss	any	individual
c<= k and c>ss	0	dynamic segmentation
c<k and c<ss	0	two-phase
c<k	> 0	static segmentation

fs: file system framework

- Handles all file-system related operation
  - Interfaces have mostly collective notion
- Interface:
  - `open()`
  - `close()`
  - `delete()`
  - `sync()`

- Current Lustre and PVFS2 fs components allow to modify stripe size, stripe depth and I/O servers used

Current status



Performance results: Tile I/O

Shark cluster at University of Houston (PVFS2):

No. of procs.	Tile Size	fcoll module	OMPIO bandwidth	ROMIO bandwidth
81	64 Bytes	Two-phase	591 MB/s	303 MB/s
81	1 MB	Dynamic Segm.	625 MB/s	290 MB/s

Deimos cluster at TU Dresden (Lustre):

No. of procs.	Tile Size	fcoll module	OMPIO bandwidth	ROMIO bandwidth
256	64 Bytes	Two-phase	2167 MB/s	411 MB/s
256	1 MB	Dynamic Segm.	2491 MB/s	517 MB/s

Tuning parallel I/O performance

- OTPO (Open Tool for Parameter Optimization): optimize the Open MPI parameter space for a particular benchmark and/or application

- Tuning for Latency I/O benchmark on shark/PVFS2
  - Parameters tuned: collective module used, number of aggregators used, cycle buffer size
- 64 different parameter combinations evaluated
- 2 parameter combinations were determined to lead to best performance:
  - *dynamic segm., 20 aggregators, 32 MB cycle buffer size*
  - *static segm. 20 aggregators, 32 MB cycle buffer size*

sharedfp framework

- Focuses around the management of a shared file pointer
  - Using a separate file and locking
  - Additional process (e.g. mpirun?)
  - Separate files per processes + metadata
  - Shared memory segment
- Collective shared filepointer operations mapped to regular collective I/O operations
- Decision logic based on
  - Location of processes
  - Availability of features (e.g. locking)
  - Hints by the user

Current status (II)

- Code committed to Open MPI repository in August 2011
- Will be part of the 1.7 release series

- Missing MPI level functionality:
  - Split collective operations (*)
  - Shared file pointer operations: developed in a separate library, currently being integrated with OMPIO (*)
  - Non-blocking individual I/O
  - Atomic access mode

Part III: Research topics

OMPIO Optimizations

- Automated selection logic for collective I/O modules
- Optimization of collective I/O operations
  - Development of new communication-optimized collective I/O algorithms (dynamic segmentation, static segmentation)
  - Automated setting of number of aggregators for collective I/O operations
  - Optimizing process placement based on I/O access pattern

- Non-blocking collective I/O operations
- Multi-thread I/O operations

Optimizing communication in collective I/O operations

File layout

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Process 0 Process 1 Process 2 Process 3

Two-phase I/O with 2 aggregators

Process 0

1	2	3	4	5	6	7	8

Process 2

9	10	11	12	13	14	15	16

Dynamic segmentation algorithm with 2 aggregators

Process 0

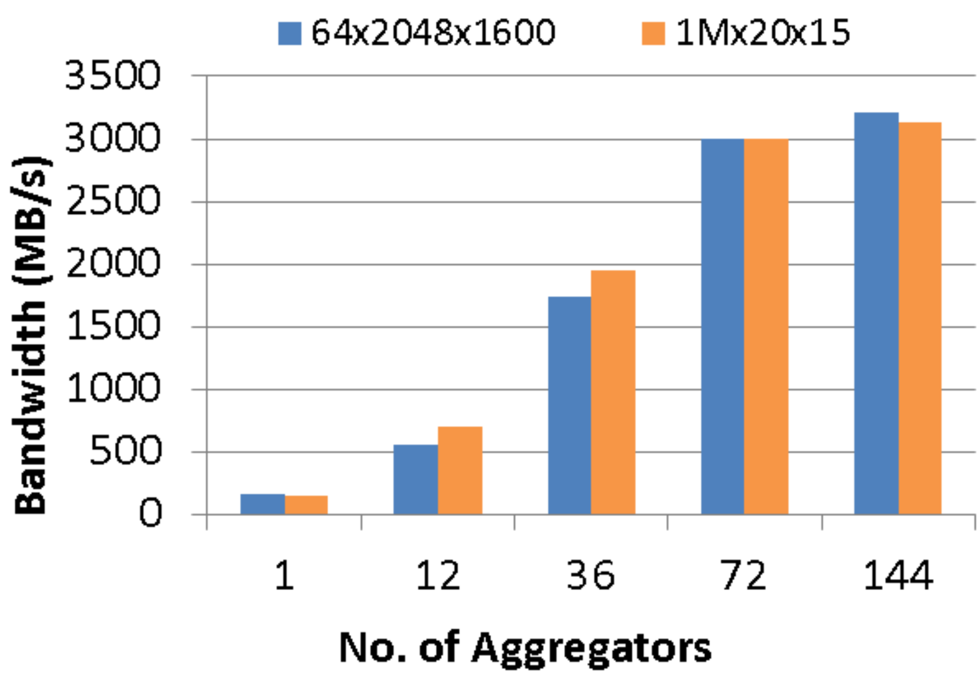
1	2	3	4
9	10	11	12

Process 2

5	6	7	8
13	14	15	16

Automated setting no. of aggregators

- No. of aggregators has enormous influence on performance, e.g.
  - Tile I/O benchmark using two-phase I/O, 144 processes, Lustre file system



Performance considerations

- Contradicting goals:
  - Generate large consecutive chunks
    -> fewer aggregators
  - Increase throughput
    -> more aggregators

- Setting number of aggregators
  - Fixed number: 1, number of processes, number of nodes, number of I/O servers
  - Tune for a particular platform and application

Determining the number of aggregators

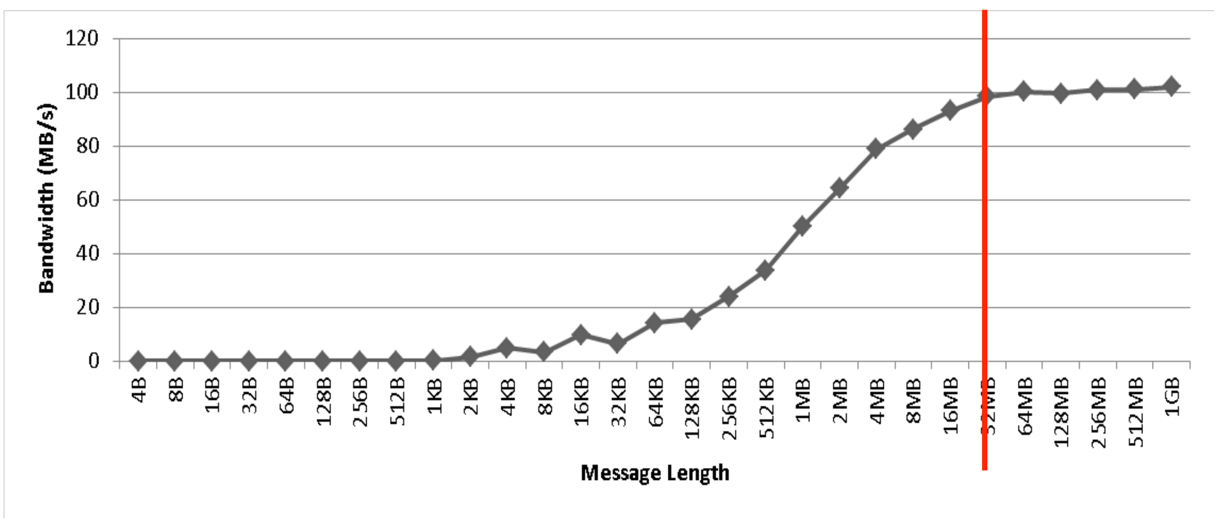
1) Determine the minimum data size *k* for an individual process which leads to maximum write bandwidth

2) Determine initial number of aggregators taking file view and/or process topology into account.

3) Refine the number of aggregators based on the overall amount of data written in the collective call

1. Determining the saturation point

- Loop of individual write operations with increasing data size
  - Avoid caching effects
  - `MPI_File_write()` vs. POSIX `write()`
  - Performed once, e.g. by system administrator
- Saturation point: first element which achieves (close to) maximum bandwidth



2. Initial assignment of aggregators

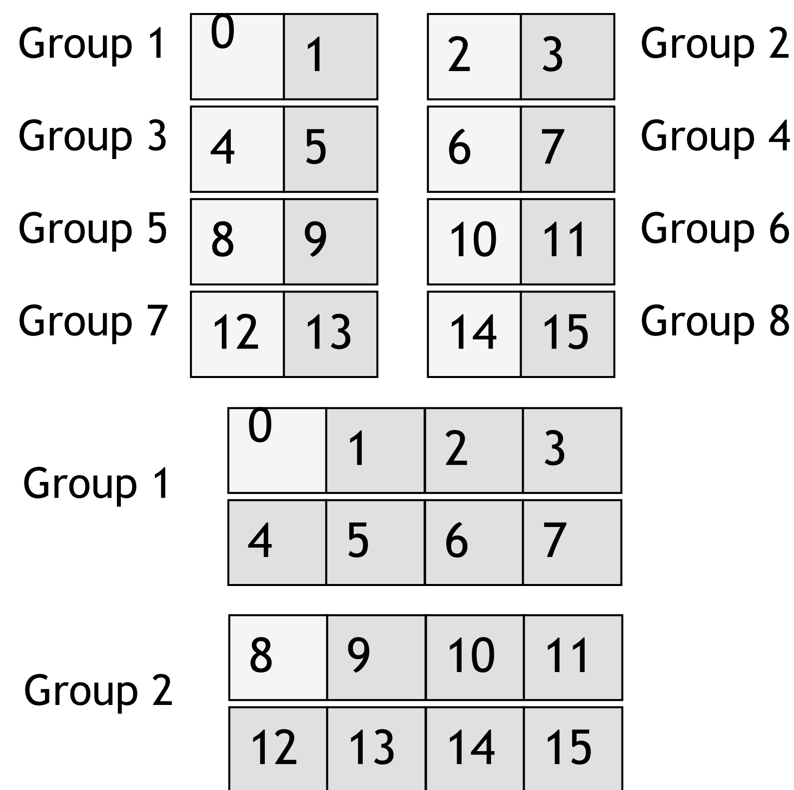
Group 1	0	1	2	3
Group 2	4	5	6	7
Group 3	8	9	10	11
Group 4	12	13	14	15

- Based on fileview
  - Based on 2-D access pattern
  - 1 aggregator per row of processes
- Based on Cartesian process topology
  - Assumption: process topology related to file access
- Based on hints
  - Not implemented at this time

- Without fileview or Cartesian topology:
  - Every process is an aggregator

3. Refinement step

- Based on actual amount of data written across all processes in one collective call
- k < no. of bytes written in group
  -> split group
- k > no. of bytes written in group
  -> merge groups



Discussion of algorithm

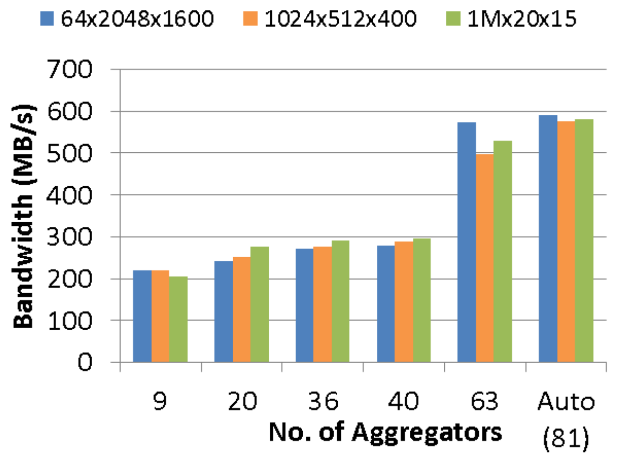
- Number of aggregators depends on overall data volume being written
  - Different calls to `MPI_File_write_all` with different data volumes will result in different number of aggregators used

- For fixed problem size, number of aggregators is independent of the number of processes used

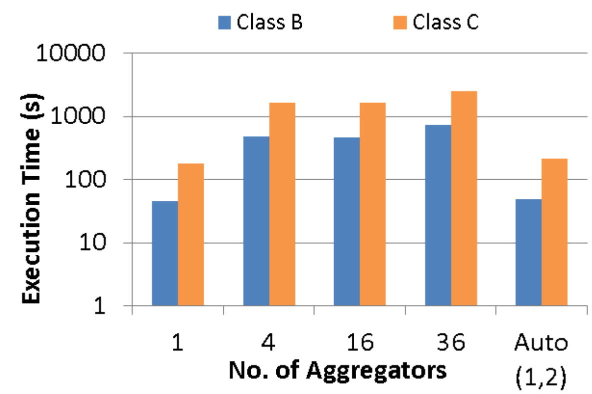
- Approach usable for two-phase I/O and some of its variants (e.g. dynamic segmentation)

Results

Tile I/O, PVFS2@shark,
81 processes, two-phase I/O



BT I/O, Lustre@deimos,
36 processes, dynamic segmentation



- 134 tests executed in total with 4 different benchmarks
  - 88 tests lead to best or within 10% of optimal performance, 110 within 25% of best performance
- Focusing on two-phase I/O algorithm only:
  - 29 out of 45 test cases outperformed one aggregator per node strategy on average by 41% (default setting by ROMIO)

I/O Access based Process Placement

- Goal: optimized placement of processes to minimize I/O time
- Three required components
  - **Application Matrix:** contains communication volumes between each pair of processes based on the I/O access pattern
  - **Architecture Matrix**: contains communication costs (bandwidth, latency) between each pair of nodes/cores
  - **Mapping Algorithm:** how to map application processes to underlying node architecture such that communication cost are minimized

Application Matrix

- Goal: predict communication occurring in collective I/O algorithm based on the access pattern of the application
- General case:
  - OMPIO extended to dump the order on how processes access the file
  - Assumption: processes which access neighboring parts of the file will have to communicate with the same aggregators
- Special case:
  - Regular access pattern (e.g. 2D data distribution and process topology)
  - Dynamic segmentation algorithm used for collective I/O
  - Communication occurs only within the outer dimension of the process topology

Application Matrix

- Simple Example : 4 processes with 2x2 tiles, each 4 bytes
- Generic Case: The file layout

1	2	1	2	1	2	1	2	3	4	3	4	3	4	3	4

0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64

- Translates to :

0	7	0	0
7	0	1	0
0	1	0	7
0	0	7	0

- Special Case : Can be represented by topology 2x2 in this case
- Which translates to :

100	100	0	0
100	100	0	0
0	0	100	100
0	0	100	100

Mapping Algorithms

- Any algorithm from literature could be used
- MPIPP Process Placement Algorithm [1]
  - Randomized algorithm based on Heuristic to exchange processes and calculate gain
  - Generic can support any kind of application and topology matrix
  - Expensive for larger number of processes
- New SetMatch Algorithm for the special case:
  - Create independent sets and matches the sets
  - Very quick even for larger number of processes
  - Greedy approach, and works for specific scenarios
  - Can be generalized by having a clustering algorithm to split sets

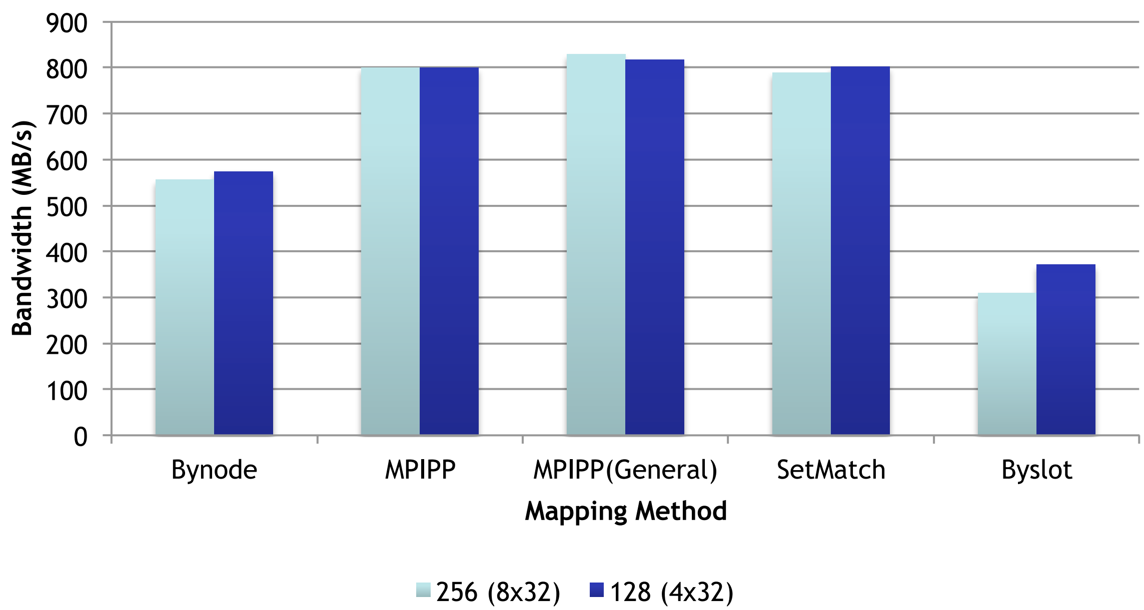
[1] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. 2006. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *Proceedings of the 20th annual international conference on Supercomputing* (ICS '06).

Preliminary Results

- Crill cluster at the University of Houston
  - Distributed PVFS2 file system using with 16 I/O servers
  - 4x SDR InfiniBand message passing network (2 ports per node)
  - 4x SDR Infiniband (1 port) I/O network
  - 18 nodes, 864 compute cores
- Focusing on collective write operations
- Modified OpenMPI trunk rev. 26077
  - Added a new rmaps component
  - Extensions to OMPIO component to extract fileview information

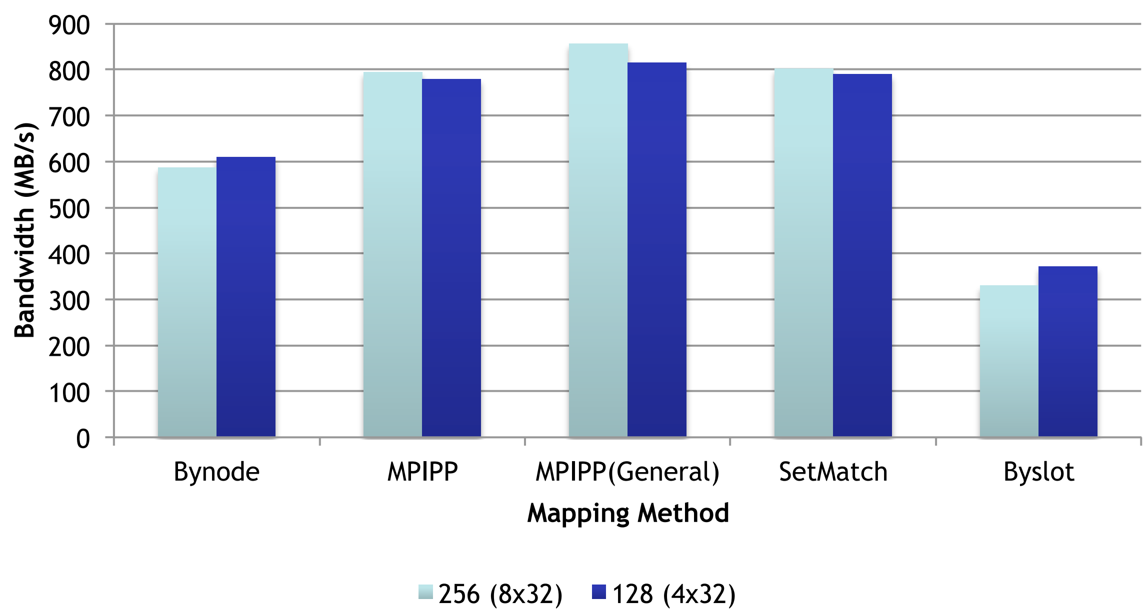
Tile I/O Results

- Benchmark : Tile I/O
- Tile Size – 1KB
- File size - 128 processes – 75G, 256 processes - 150G



Tile I/O Results - II

- Benchmark : Tile I/O
- Tile Size – 1MB
- File size - 128 processes – 75G, 256 processes - 150G



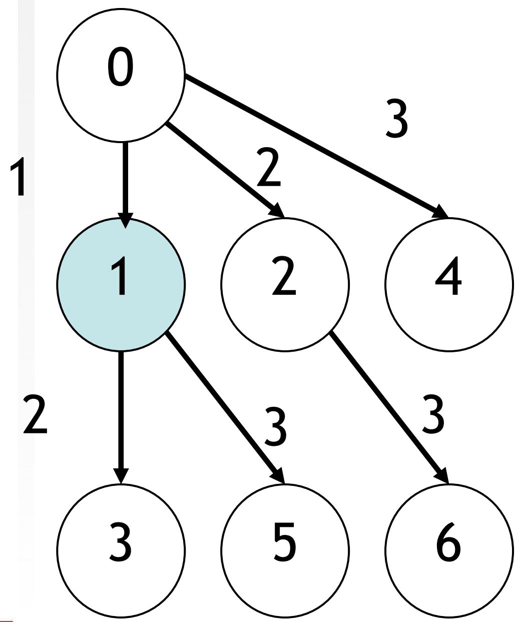
Non-blocking collective operations

- Non-blocking collective Operations
  - Hide communication latency by overlapping
  - Better usage of available bandwidth
  - Avoid detrimental effects of pseudo-synchronization
  - Demonstrated benefits for a number of applications
- Was supposed to be part of the MPI-3 specification
  - Passed 1st vote, failed in 2nd vote

Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI, Supercomputing 2007.

Overview of LibNBC

- Implements non-blocking versions of all MPI collective operations
- Schedule based design: a process-local schedule of p2p operations is created



Pseudocode for schedule at rank 1:

```
NBC_Sched_recv(buf, cnt, dt, 0, sched);
NBC_Sched_barr(sched);
NBC_Sched_send(buf, cnt, dt, 3, sched);
NBC_Sched_barr(sched);
NBC_Sched_send(buf, cnt, dt, 5, sched);
```

See http://www.unixer.de/publications/img/hoefler-hlrs-nbc.pdf for more details

Overview of LibNBC

- Schedule execution is represented as a state machine
- State and schedule are attached to every request
- Schedules might be cached/reused

- Progress is most important for efficient overlap
  - Progression in `NBC_Test/NBC_Wait`

Collective I/O operations

- Collective operation for reading/writing data allows to combine data of multiple processes and optimize disk-access
- Most popular algorithm: two-phase I/O
- Algorithm for a collective write operation
  - Step 1:
    - gather data from multiple processes on aggregators
    - Sort data based on the offset in the file
  - Step 2: aggregators write data

Nonblocking collective I/O operations

```
MPI_File_iwrite_all (MPI_File file,
    void *buf, int cnt, MPI_Datatyep dt,
    MPI_Request *request);
```

- Difference to nonblocking collective communication operations:
  - Every process is allowed to provide different amounts of data per collective read/write operation
  - No process has a ‘global’ view how much data is read/written

Nonblocking collective I/O operations

- Total amount of data necessary to determine
  - How many cycles are required
  - How much data a process has to contribute in each cycle

⇨ schedule for libNBC can not be constructed in `MPI_File_iwrite_all`

- Further consequence:
  - some temporary buffer required internally by the algorithm can not be allocated when posting the operation

Nonblocking collective I/O operations

- Create a schedule for a non-blocking Allgather(v)
  - Determine the overall amount of data written across all processes
  - Determine the offsets for each data item within each group

- Upon completion:
  - Create a new schedule for the shuffle and I/O steps
  - Schedule can consist of multiple cycles

Extensions to libNBC

- New internal libNBC operations for:
  - Non-blocking read/write operation
  - Compute operations for sorting and merging entries
  - Buffer management (allocating, freeing buffers)
  - New nonblocking send/recv primitives with additional level of buffer indirections for dynamically allocated buffers
- Progressing multiple, different types of requests simultaneously

PARALLEL SOFTWARE TECHNOLOGIES LABORATORY PSTL

Caching of schedules

- Very difficult for I/O operations
  - Subsequent calls to `MPI_File_iwrite_all` will have different offsets into the file
    - Amount of data provided by a process in a cycle depends on the offset in the file
  - Processes allowed to mix individual and collective I/O calls

⇨ Not possible to predict offsets of other processes and to reuse a schedule

Caching of schedules (II)

- When using different files
  - offsets might be the same across multiple function calls, but different file handles will be used
  - Caching typically done on communicator / file handle

⇨ Caching across different file handles difficult, but no impossible

Experimental evaluation

- Crill cluster at the University of Houston
  - Distributed PVFS2 file system using with 16 I/O servers
  - 4x SDR InfiniBand message passing network (2 ports per node)
  - Gigabit Ethernet I/O network
  - 18 nodes, 864 compute cores
- LibNBC integrated with OpenMPI trunk rev. 24640
- Focusing on collective write operations

Latency I/O tests

- Comparison of blocking and nonblocking versions
  - No overlap
  - Writing 1000 MB per process
  - 32 aggregator processes, 4MB cycle buffer size
  - Average of 3 runs

No. of processes	Blocking Bandwidth [MB/s]	Non-blocking bandwidth [MB/s]
64	703	660
128	574	577

Latency I/O overlap tests

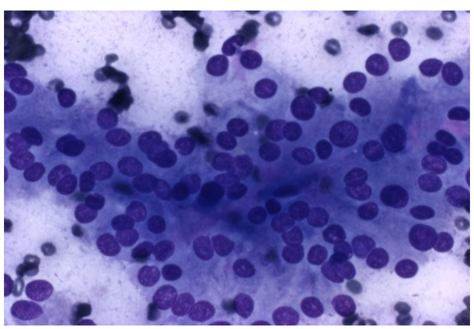
- Overlapping nonblocking coll. I/O operation with equally expensive compute operation
  - Best case: overall time = max (I/O time, compute time)

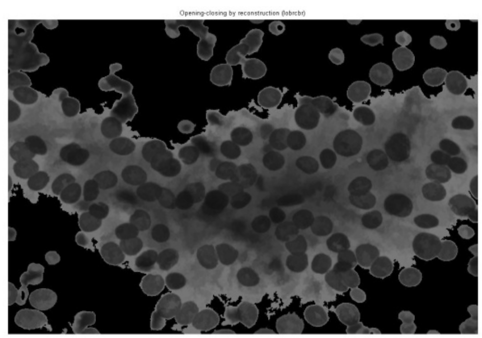
- Strong dependence on ability to make progress
  - Best case: time between subsequent calls to `NBC_Test` = time to execute one cycle of coll. I/O

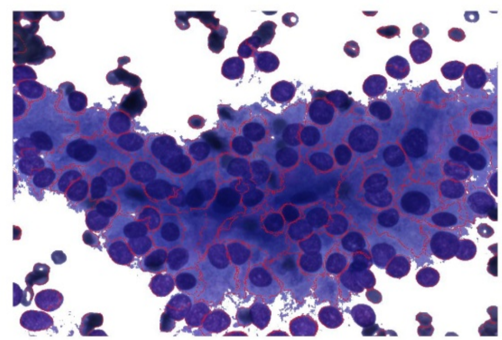
No. of processes	I/O time	Time spent in computation	Overall time
64	85.69 sec	85.69 sec	85.80 sec
128	205.39 sec	205.39 sec	205.91 sec

Parallel Image Processing Application

- Used to assist in diagnosing thyroid cancer
- Based on microscopic images obtained through Fine Needle Aspiration (FNA)
- Slides are large
  - typical image: 25K x 70K pixels, 3-6 Gigabytes/slide
  - multispectral imaging to analyze cytological smears







Parallel Image Processing Application

- Texture based image segmentation

  For each Gabor Filter
  - *Forward FFT of Gabor Filter*
  - *Convolution operation of Filter and Image*
  - *Backward FFT of the convolution result*
  - *Optionally: write result of backward FFT to file*

- FFT operations based on FFTW 2.1.5

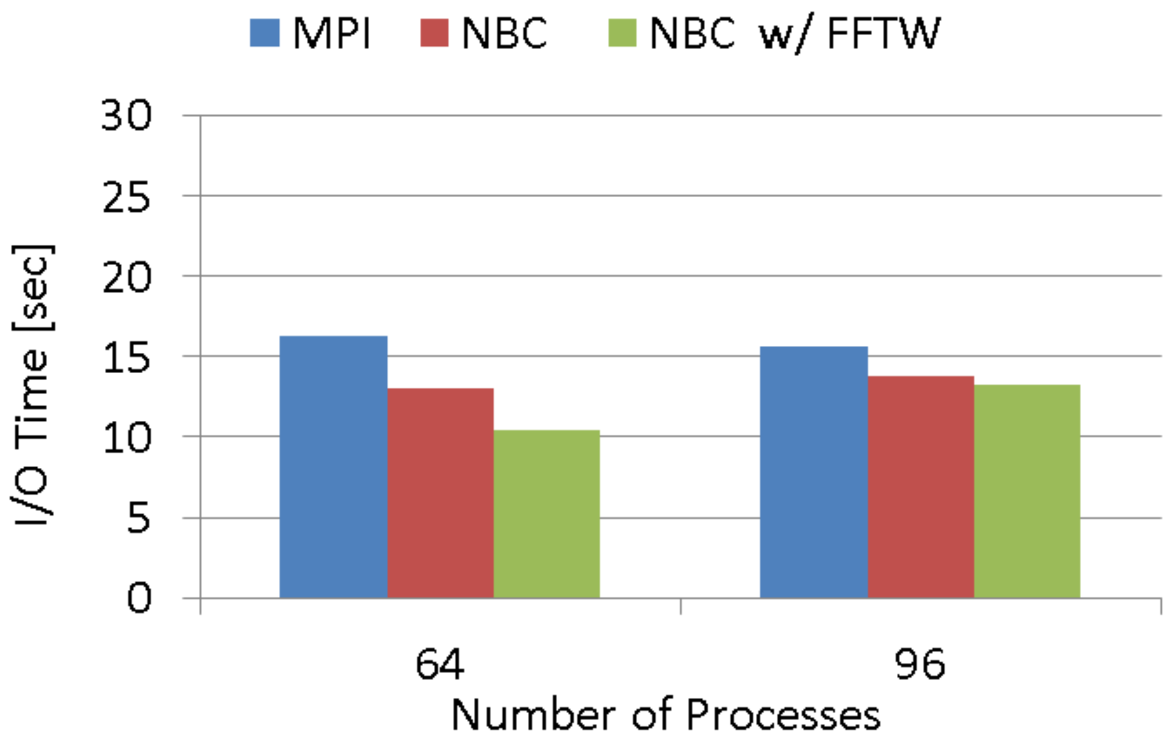
Parallel Image Processing Application

- Code modified to overlap write of iteration *i* with computations of iteration *i+1*

- Two code versions generated:
  - ***NBC***: Additional calls to progress engine added between different code blocks
  - ***NBC w/FFTW:*** Modified FFTW to insert further calls to progress engine

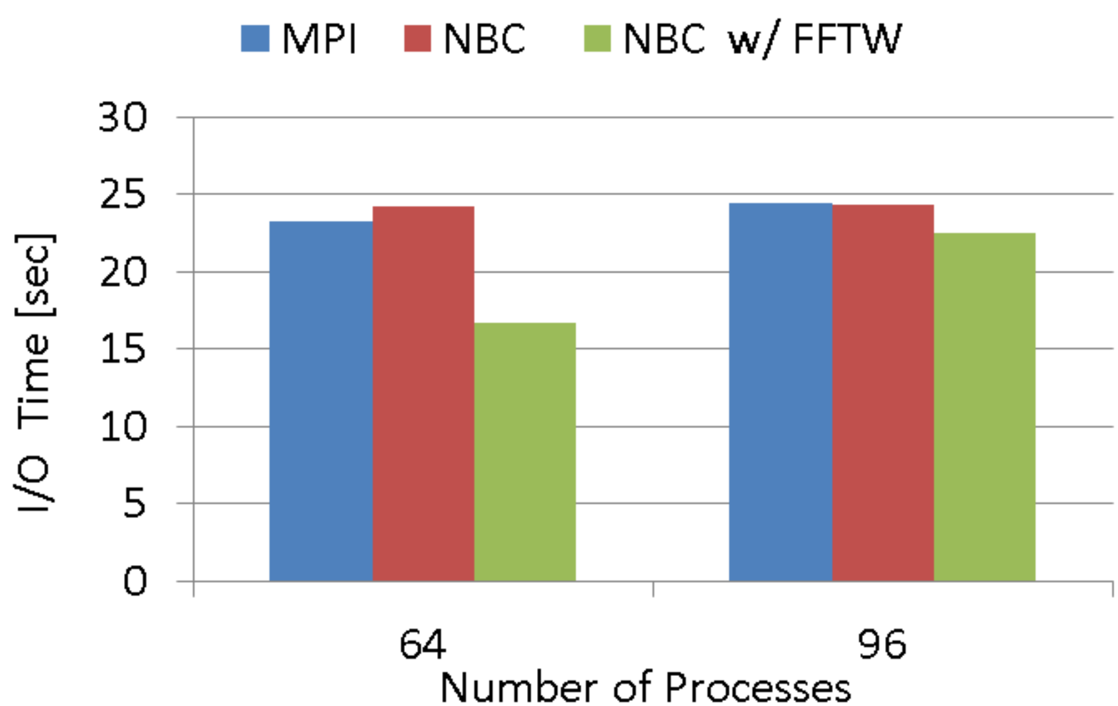
Application Results (I)

- 8192 x 8192 pixels, 21 spectral channels
- 1.3 GB input data, ~3 GB output data
- 32 aggregators with 4 MB cycle buffer size



Application Results (II)

- 12281 x 12281 pixels, 21 spectral channels
- 2.95 GB input data, ~7 GB output data
- 32 aggregators with 4 MB cycle buffer size



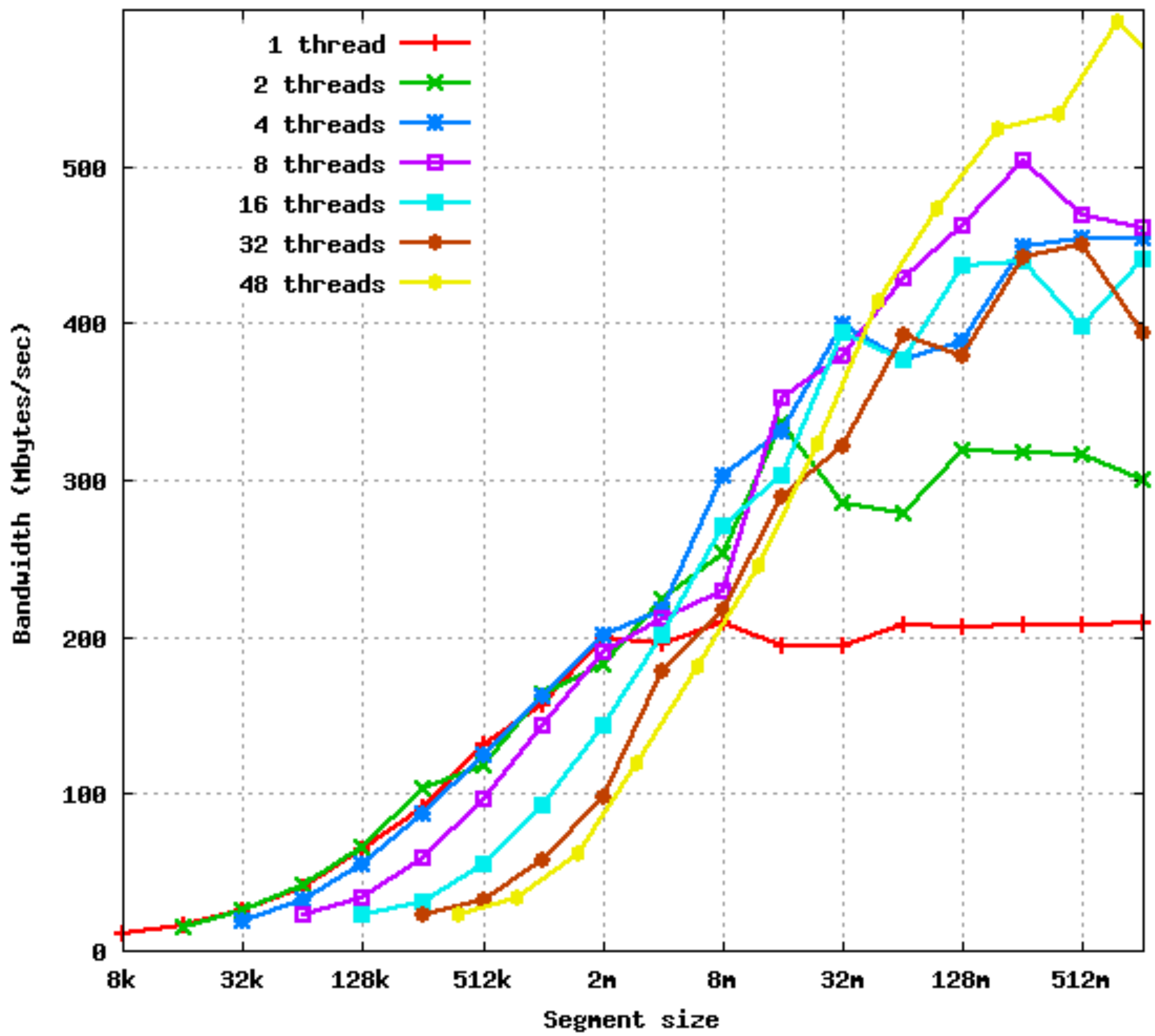
Multi-threaded I/O optimization

- Currently no support for parallel I/O in OpenMP
- Need for threads to be able to read/write to the same file
  - Without locking file handle
  - Without having to write to separate files to obtain higher bandwidth
  - Applicable for all languages supported by OpenMP
- API specification:
  - All routines are library functions (not directives)
  - Routines implemented as collective functions
  - Shared file pointer between threads
  - Support for List I/O Interfaces

Overview of Interfaces (*write*)

File Manipulation		omp_file_open_all
		omp_file_close_all
Different Arguments	Regular I/O	omp_file_write_all
		omp_file_write_at_all
	List I/O	omp_file_write_list_all
		omp_file_write_list_at_all
Common arguments	Regular I/O	omp_file_write_com_all
		omp_file_write_com_at_all
	List I/O	omp_file_write_com_list_all
		omp_file_write_com_list_at_all

Results – *omp_file_write_all*



Performance Results

- OpenMP version of the NAS BT Benchmark
- Extended to include I/O operations

No. of Threads	PVFS2 [sec]	PVFS2-SSD [sec]
1	410	691
2	305	580
4	168	386
8	164	368
16	176	368
32	172	368
48	168	367

Summary and Conclusions

- I/O is one of the major challenges for current and upcoming high-end systems
- Huge potential for performance improvements
- OMPIO provides a highly modular architecture for parallel I/O

- To improve out-of-the-box performance of I/O libraries
  - Algorithmic developments necessary
  - Handling fat multi-core nodes still a challenge

Contributors

- Vishwanath Venkatesan
- Kshitij Mehta
- Carlos Vanegas

- Mohamad Chaarawi
- Ketan Kulkarni
- Suneet Chandok

- Rainer Keller (University of Applied Sciences Stuttgart)